

Junho de
2022

Documentação Sistema de Escambo - Sistema Distribuidos

Samuel Silva Costa Nascimento

2662

Pedro Henrique Silva Oliveira

2677



UFV

Universidade Federal de Viçosa

Sumário

03	—	Introdução
04	—	Requisitos Funcionais
05	—	Casos de Uso
05	—	Diagrama de Caso de Uso
06	—	Modelo Físico
06 - 09	—	Modelo Arquitetural
10 - 11	—	Implementação Socket
11	—	Conclusão parte III
12 - 21	—	Comunicação/Requisição & Resposta
22 - 23	—	Dados para teste
24 - 26	—	Troca da comunicação de Socket para GRPC
27	—	Conclusão

Introdução

O projeto consiste na implementação de um sistema distribuído de trocas de pokemóns. O Sistema vai permitir que cada jogador, ao se cadastrar, obtenha uma conta com 10 pokemóns aleatórios iniciais que serão utilizadas para troca. Cada jogador possui seu inventário que pode ser visualizado por ele e por outros usuários para as trocas e, dessa forma, encontrem os pokemóns necessários para compor o seu time. Ao encontrar um ou mais pokemóns que deseja no inventário de outro, o jogador pode fazer um pedido de troca, selecionando os pokemóns que deseja em troca dos seus. O usuário poderá visualizar todos os pedidos de troca recebidos e enviados e, dessa forma, decidir se aceita ou não as trocas recebidas e, também, acompanhar suas trocas solicitadas.

Requisitos Funcionais

[RF01] - O sistema deve permitir a criação de um jogador.

[RF02] - O sistema deve permitir o login de um jogador.

[RF03] - O sistema deve permitir a um jogador enviar um pedido de troca para outro jogador.

[RF04] - O sistema deve permitir a um jogador aceitar um pedido de troca de outro jogador.

[RF05] - O sistema deve permitir a um jogador rejeitar um pedido de troca de outro jogador.

[RF06] - O sistema deve permitir a um jogador visualizar os pokemons de outros jogadores.

[RF07] - O sistema deve permitir a um jogador inserir pokemons para troca.

[RF08] - O sistema deve permitir a um jogador remover pokemons para troca.

[RF09] - O sistema deve permitir a um jogador realizar a troca.

[RF10] - O sistema deve permitir a um jogador visualizar o seu inventário de pokemons.

Casos de Uso

1. Realizar Cadastro.
2. Efetuar Login.
3. Gerir inventário.
4. Gerir pedidos de troca.
5. Visualizar pokemons de outros jogadores.
6. Visualizar o próprio inventário.

Diagrama de Casos de Uso

Diagrama de Caso de uso



Requisitos Funcionais

- [RF01] - O sistema deve permitir a criação de um jogador
 [RF02] - O sistema deve permitir o login de um jogador
 [RF03] - O sistema deve permitir a um jogador enviar um pedido de troca para outro jogador
 [RF04] - O sistema deve permitir a um jogador aceitar um pedido de troca de outro jogador
 [RF05] - O sistema deve permitir a um jogador rejeitar um pedido de troca de outro jogador
 [RF06] - O sistema deve permitir a um jogador visualizar os pokemons de outros jogadores
 [RF07] - O sistema deve permitir a um jogador inserir pokemons para troca
 [RF08] - O sistema deve permitir a um jogador remover pokemons para troca.
 [RF09] - O sistema deve permitir a um jogador realizar a troca.
 [RF10] - O sistema deve permitir a um jogador visualizar o seu inventário de pokemons.

Casos de Uso

1. Realizar Cadastro.
2. Efetuar Login.
3. Gerir inventário.
4. Gerir pedidos de troca.
5. Visualizar pokemons de outros jogadores.
6. Visualizar o próprio inventário.

Modelo Físico

O Sistema de escambo a ser desenvolvido, por se tratar de uma aplicação simples e objetiva se encaixa no modelo físico **"Early"**, onde a ideia é ser um produto fechado, com pouca escalabilidade, uma qualidade de serviço básica, com heterogeneidade limitada e nenhuma prioridade de estar aberto a grandes comunicações externas.

Modelo Arquitetural

Os Sistemas distribuídos são sistemas complexos devido ao fato de conter diversos processos separados em diversas máquinas e que se comunicam e se conectam através de rede de computadores. A partir disso é que antes de iniciar a produção do Sistema de Escambo de Pokemons, um modelo arquitetural foi definido e construído.

Iniciando pelas entidades arquitetônicas, do ponto de vista do sistema:

- Processos (Threads)

Do ponto de vista da programação:

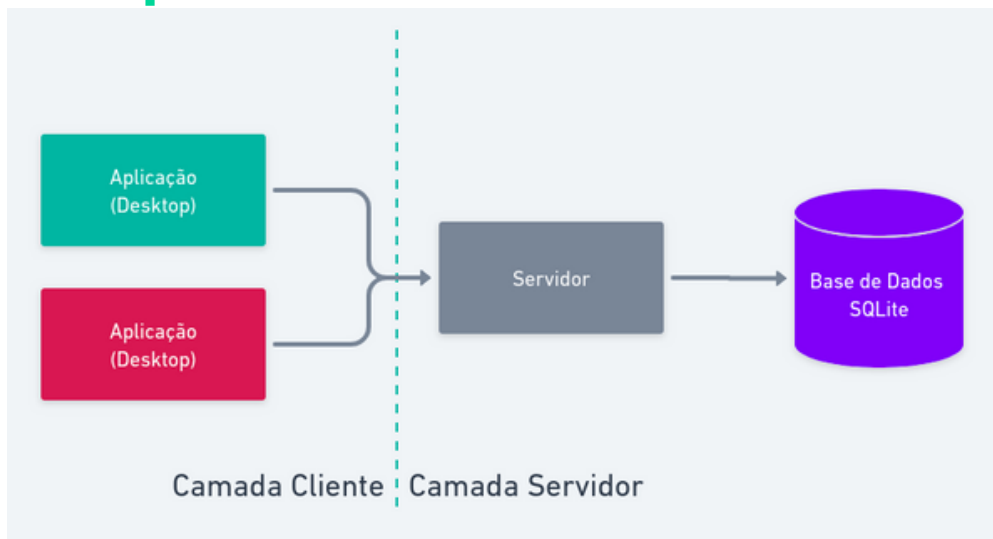
- Objetos (Acessados por interfaces)

O Paradigma de comunicação que mais se encaixa é a comunicação entre processos.

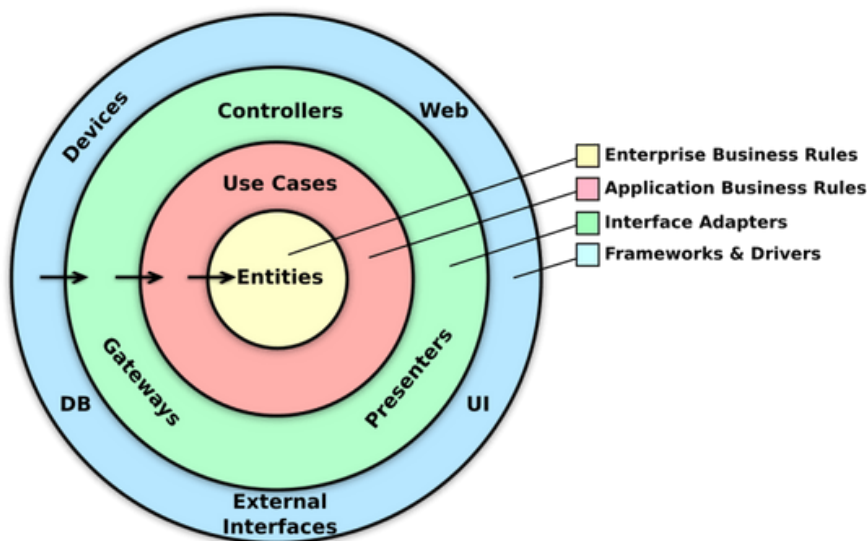
O modelo de Arquitetura básico escolhido para construção foi o de Cliente-Servidor, onde nossos processos ao se interagirem vão assumir determinadas funções, sendo um deles cliente e o outro servidor. O cliente será responsável apenas por receber as entradas do usuário e permitir o mesmo realizar requisições ao servidor para realizar trocas, consultar o inventário de outros

usuários, recusar/aceitar pedidos de troca e etc. O servidor será responsável por receber essas requisições e, de acordo com os parâmetros e rotas utilizados, proverá as informações e fará a persistência dos dados. O sistema distribuído vai ser estruturado inicialmente em duas camadas físicas, onde temos apenas o client-side e o server-side, porém posteriormente a ideia é que ele se adeque a estrutura em três camadas. E a nível de estrutura de sistema o nosso SD vai ser estruturado em camada lógica utilizando a Arquitetura Limpa.

Arquitetura – camada física



Arquitetura – camada lógica



A comunicação entre SD é um foco importante a se tratar, devido ao fato de que os processos se comunicam através de troca de mensagens, por isso o modelo de interações procura analisar e tratar qualquer tipo de falha relacionado a esse processo. A Latência, taxa de transmissão de dados e jitter, são problemas que ocorrem na comunicação entre as entidades, então a preocupação em como o sistema distribuído vai lidar com todas essas questões é extremamente importante. O problema de latência se manifesta quando temos atraso na entrega de uma mensagem, o que pode ocasionar diversos outros problemas na funcionalidade de um dos processos. Já a taxa de transmissão é dado como a quantidade disponível de banda que um processo possui para otimizar a quantidade de dados a serem transmitidos na mensagem. E por fim o Jitter é quando temos uma oscilação no tempo de transmissão de pacotes. Todos esses pontos supracitados são importantes de serem analisados e definidos métodos resolutivos para o tratamento desses casos que ocorra no Sistema Distribuído desenvolvido.

O Sistema Distribuído a ser desenvolvido utilizará o modelo assíncrono, já que vamos trabalhar em um ambiente não controlado, onde pode ter oscilação entre o tempo de execução das funcionalidades de cada processo, a comunicação entre os processos não vai ser controlada podendo ocorrer em tempos diferentes e não se importará com o Drift rate dos processos.

O modelo de falhas também é um dos fatores a ser observado durante o processo de construção da arquitetura do sistema. Dentre os modelos de falhas, a falha arbitrária seria uma das possíveis a acontecer durante a execução do SD, já que um dos processos pode simplesmente falhar e encerrar sua execução. Uma forma de se evitar com que esse tipo de falha aconteça é priorizando a confiabilidade do sistema. Outra falha se dá na comunicação entre processos pois uma requisição pode falhar devido a inúmeros problemas, logo também é necessário uma confirmação do servidor que é recebida pelo cliente com a finalidade de tornar o sistema mais responsivo, deixando claro ao usuário o que está acontecendo a cada requisição para o caso de falhas que impossibilitam a continuidade da execução de apenas alguns componentes do sistema.

Segurança é um aspecto importante de ser observado e implementado em um SD, porém é algo que tem um custo muito alto no quesito processamento, sendo assim como o sistema produzido não possui grandes funcionalidades, utilizar algum modelo de segurança seria gastos desnecessários.

Implementação Socket

Para a implementação do nosso sistema de escambo o PokeTroca, foi utilizado da API de sockets como o meio de comunicação entre os processos Cliente e Servidor, através da troca biletarel de bytes, escolhemos a representação externa de dados Json como meio de padronizar toda a estrutura de obtenção dos dados. Portanto a comunicação entre o cliente servidor se dá através de sockets e todas as mensagens enviadas e recebidas estão em JSON. Iniciamos o projeto pensando na maneira de se estruturar logicamente o servidor para que a camada de negócio ficasse totalmente desacoplada das camada de infraestrutura, ou seja, mesmo que tenhamos que trocar a forma de comunicação, por exemplo, API de Sockets para o Midleeware GRPC não seria necessário uma mudança muito grande na estrutura do código. Essa maneira de estruturação lógica, é denominada como Arquitetura Limpa, onde trabalhamos dividindo o processo do servidor nas camadas, domain, presentation, data, infra e main. A primeira dificuldade então foi conseguir fazer com que toda essa arquitetura lógica funcionasse da maneira mais adequada, além de que a linguagem escolhida (python) estava fora da nossa zona de conforto. Após a definição e a construção da estrutura, decidimos separar os papeis, onde o integrante Samuel ficou responsável pela implementação do Servidor e o integrante Pedro ficou responsável pela implementação do Cliente, utilizando o framework Kivy para o desenvolvimento da interface gráfica.

A segunda dificuldade foi na utilização do framework Kivy, que assim como outras bbts nativas da linguagem python para produção de interface grafica possui uma curva de aprendizado alta, portanto em algumas partes do funcionamento do sistema é possível que se encontre alguns fix de layouts, mas que não prejudicam o funcionamento do sistema. A terceira dificuldade encontrada foi no processo de comunicação, pois além de padronizar as mensagens utilizado o JSON, tivemos que aumentar o tamanho do buffer para permitir com que a resposta do processo servidor enviasse uma quantidade maior de dados. Enfim, as questões supracitadas foram as dificuldades encontradas durante o processo de implementação do projeto.

Conclusão parte III

A terceira parte do trabalho realmente trouxe bastante desafio, desde a linguagem de programação a ser utilizada, até a estrutura de pastas. Porém a implementação de desse trabalho utilizando Sockets foi muito interessante e agregou muito em vários aspectos:

- Criação de um sistema sem a utilização de todos os middlewares facilitadores que temos atualmente.
- A utilização da orientação a objetos para construir todo o sistema.
- A divisão do trabalho em dois pedaços que se comunicavam e sem problema algum de conflito devido a fluidez da comunicação.
- Desenvolvimento de um sistema do completo zero saindo totalmente da zona de conforto que é o desenvolvimento web.

Enfim, todas as informações supracitadas nós levaram a uma conclusão positiva da implementação dessa terceira parte.

Comunicação/Requisição & Resposta

Tela de Login

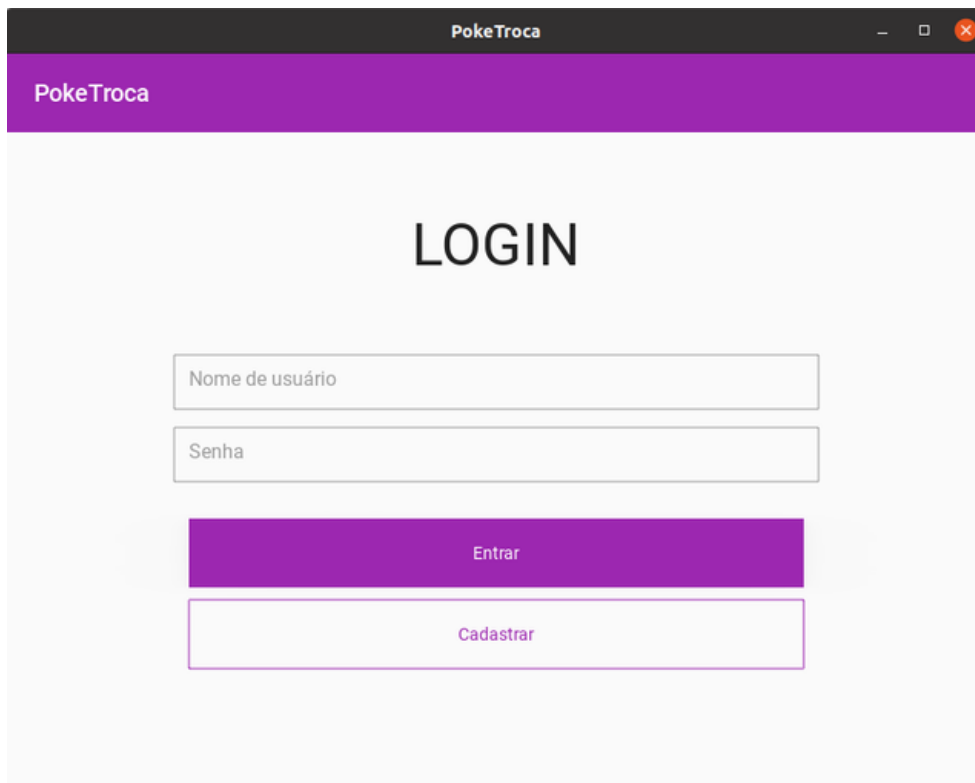
Caso de uso: Realizar Login

Esse caso de uso é responsável por pegar os dados de e-mail e senha preenchidos pelo usuário no Cliente e enviar para o Servidor, caso o usuário exista na base do SQLite e a senha estiver correta, uma resposta de sucesso é enviada para o Cliente e o usuário consegue acessar o sistema. Caso o contrário aconteça, uma mensagem de erro é mostrada ao cliente.

Requisição e Resposta

```
Request: {  
  "route": 'login',  
  "params": {  
    "email": 'testee@teste.com',  
    "password": '123teste'  
  }  
}  
  
# Caso de sucesso  
Response: {  
  "statusCode": 200,  
  body: { "id": 1, "username": 'teste', "name": 'samuel' }  
}  
  
# Caso de erro  
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }  
  
# Entidade  
user
```

Tela de Login



A imagem mostra a interface de login de uma aplicação web chamada 'PokeTroca'. No topo, há uma barra de título com o nome 'PokeTroca' e ícones de minimizar, maximizar e fechar. Abaixo, o nome 'PokeTroca' aparece novamente em uma barra de cor verde. O título principal da tela é 'LOGIN' em letras maiúsculas e cinza. Abaixo dele, há dois campos de entrada: 'Nome de usuário' e 'Senha', ambos com bordas arredondadas e um ícone de olho para alternar a visibilidade da senha. Abaixo dos campos, há dois botões: 'Entrar' em uma barra de cor verde e 'Cadastrar' em uma barra branca com uma borda verde.

Tela de Cadastro

Caso de uso: Realizar Cadastro

Esse caso de uso é responsável por pegar os dados de e-mail, senha e nome preenchidos pelo usuário no Cliente e enviar para o Servidor, caso o e-mail não exista na base do SQLite, uma resposta de sucesso é enviada para o Cliente e o usuário é cadastrado. Caso o contrário aconteça, uma mensagem de erro é mostrada ao cliente.

Requisição e Resposta

```
Request: {
  "route": 'user/create',
  "params": {
    "username": 'teste2@teste.com',
    "password": '123teste',
    "name": 'teste'
  }
}

# Caso de sucesso
Response: { "statusCode": 200, body: {} }

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'Usuario já existente!' } }

# Entidade
user
```

Tela de cadastro

The screenshot shows a mobile application interface for 'PokeTroca'. The top navigation bar is purple with a white back arrow and the text 'PokeTroca'. The main content area is light gray and features the word 'CADASTRO' in large, bold, black letters. Below this, there are three white input fields with gray borders, each with a label: 'Nome', 'Nome de usuário', and 'Senha'. At the bottom of the screen, there is a solid purple bar containing a white button labeled 'Cadastrar'.

Tela Inicial

Caso de uso: Recuperar lista de usuários

Esse caso de uso é responsável por pegar os dados de todos os usuários (menos a senha) e enviar para o Cliente, que vai mostrar na tela todos esses usuários através de uma lista.

Requisição e Resposta

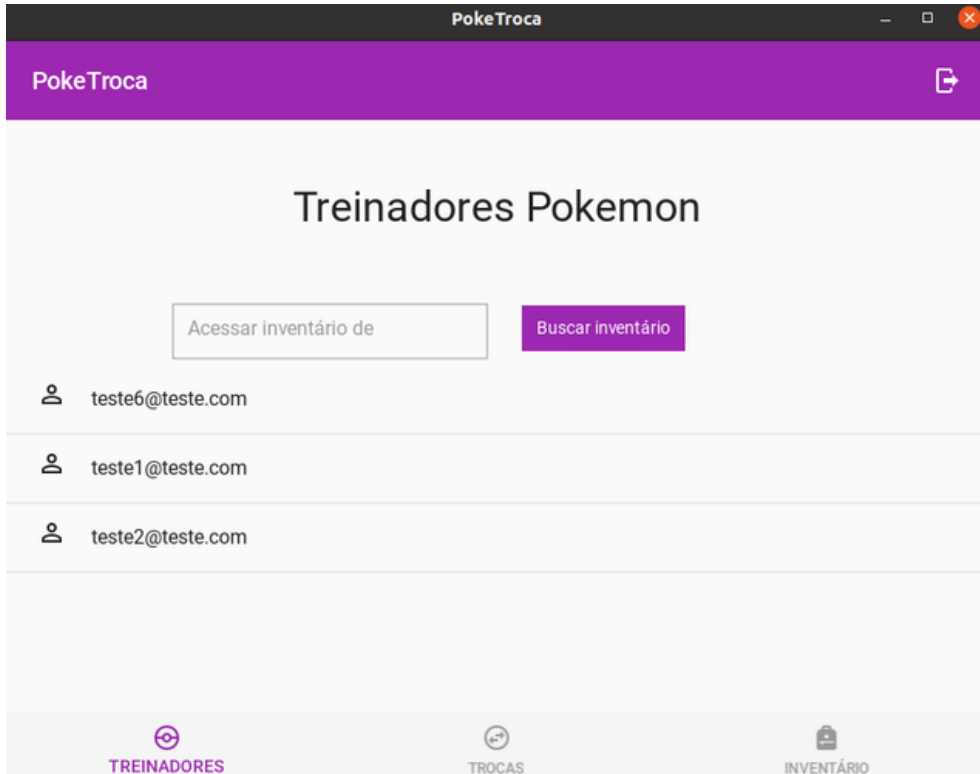
```
Request: { "route": get-user, "params": {} }

# Caso de sucesso
Response: { "statusCode": 200, body: [
  {
    "id": 1,
    "username": 'teste1@teste.com',
    "name": 'teste'
  },
  {
    "id": 2,
    "username": 'teste2@teste.com',
    "name": 'teste2'
  }
]
}

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
user
```

Tela inicial



Tela de Solicitações de troca

Caso de uso: Recuperar todas as solicitações de troca do usuário

Esse caso de uso é responsável por pegar os dados de todas as solicitações de troca que o usuário logado recebeu e enviar para o Cliente, que vai mostrar na tela todas essas solicitações através de uma lista.

Requisição e Resposta

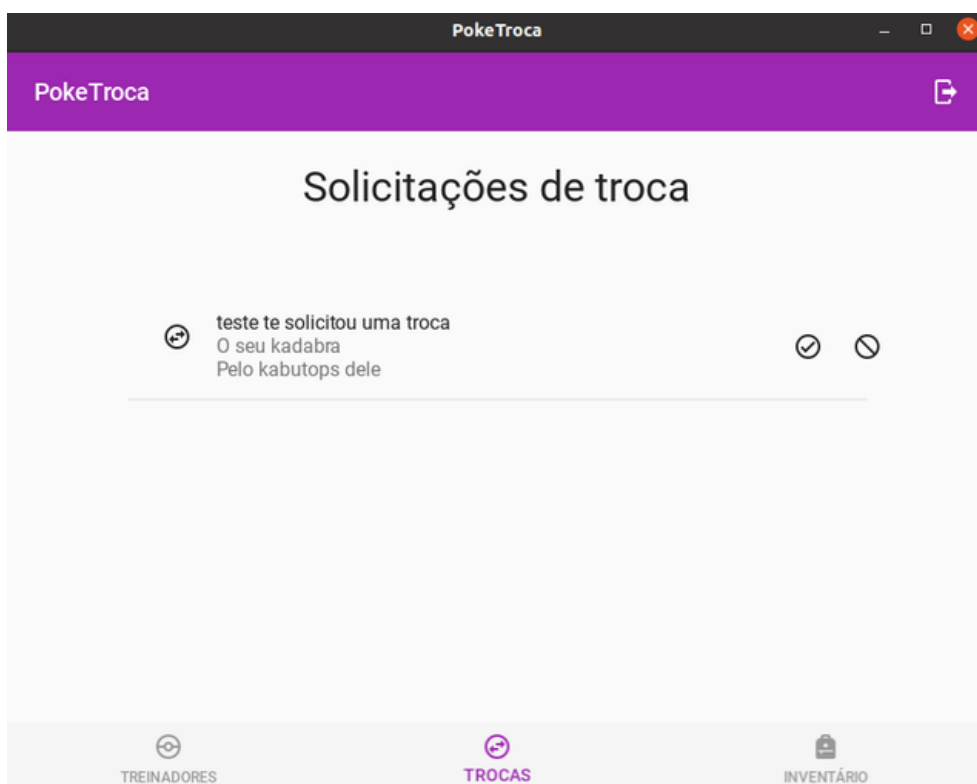
```
Request: { "route": 'trade/solicitations', "params": { "id": 2 } }

# Caso de sucesso
Response: { "statusCode": 200, body: [
  {
    "received_user_id": '',
    "sender_user_id": '',
    "want_pokemon_id": '',
    "give_pokemon_id": '',
    "status": ''
    "want_pokemon_name": ''
    "want_pokemon_image": ''
    "give_pokemon_name": ''
    "give_pokemon_image": ''
    "received_user_name": ''
    "sender_user_name": ''
  }
]
}

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
trade
```

Tela de solicitação de troca



Caso de uso: Realizar uma troca

Esse caso de uso é responsável por pegar os dados do id da solicitação e enviar para o Servidor, que vai realizar a troca dos pokemons, atualizar o status da solicitação para finalizado e enviar uma mensagem de sucesso para o Cliente.

Requisição e Resposta

```
Request: { "route": 'trade/accecpt', "params": { "id": 1 } }

# Caso de sucesso
Response: { "statusCode": 200, body: {} }

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
trade, user, inventory, inventorypokemon, pokemon
```

Caso de uso: Rejeitar uma troca

Esse caso de uso é responsável por pegar os dados do id da solicitação e enviar para o Servidor, que vai realizar a rejeição da troca dos pokemons, atualizar o status da solicitação para finalizado e enviar uma mensagem de sucesso para o Cliente.

Requisição e Resposta

```
Request: { "route": 'trade/refuse', "params": { "id": 1 } }

# Caso de sucesso
Response: { "statusCode": 200, body: {} }

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
trade|
```

Tela de Inventário

Caso de uso: Recuperar inventário de um usuário

Esse caso de uso é responsável por pegar os dados do inventário de um usuário através de um id específico passado na requisição e enviar para o Cliente, que vai mostrar na tela todos os pokemons existentes nesse inventário através de uma lista.

Requisição e Resposta

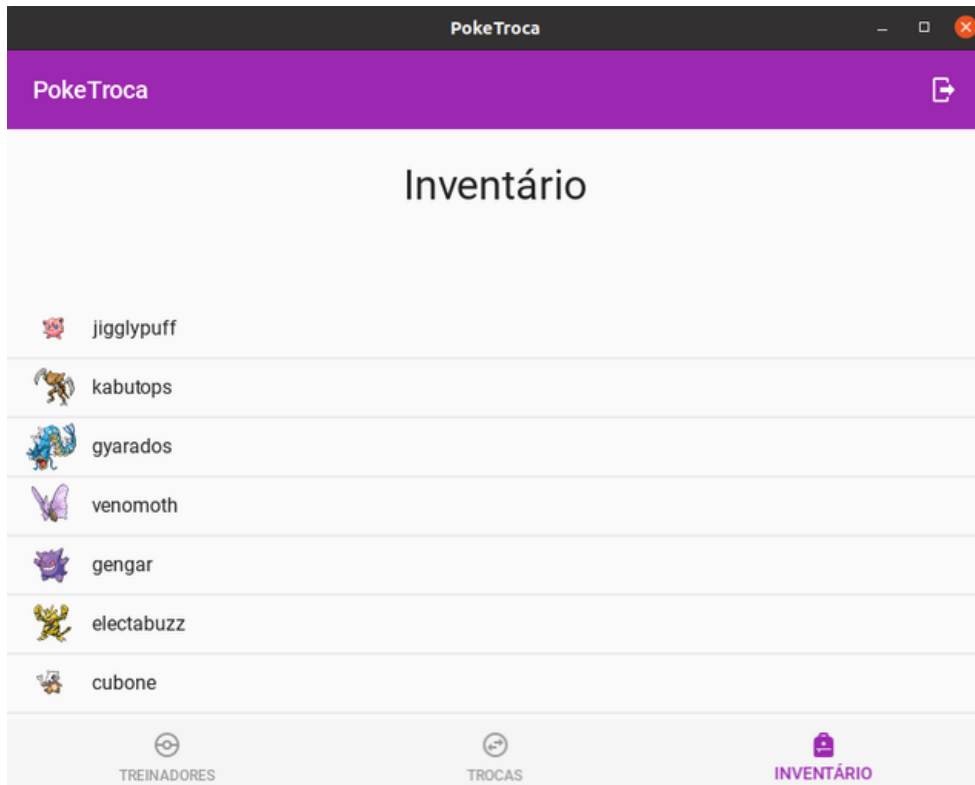
```
Request: { "route": 'user/inventory', "params": { "id": 2 } }

# Caso de sucesso
Response: { "statusCode": 200, body: {
  "user_id": 1,
  "user_name": '',
  "inventory_id": '',
  "inventory_data": [
    {
      "pokemon_id": '',
      "pokemon_name": '',
      "pokemon_image": ''
    }
  ]
}
}

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
user, inventory, inventorypokemon, pokemon
```

Tela de inventário



Tela de Troca

Caso de uso: Criar uma solicitação de troca

Esse caso de uso é responsável por pegar os dados do pokemon que o usuário deseja, os dados do pokemon que o usuário está dando em troca, os seus dados e os dados do usuário que está recebendo a troca e envia para o Servidor, que vai inserir no banco de dados a nova solicitação e enviar uma mensagem de sucesso para o usuário.

Requisição e Resposta

```
Request: { "route": 'trade/create', "params": {
  "received_user_id": 3,
  "sender_user_id": 2,
  "want_pokemon_id": 64,
  'give_pokemon_id': 32
}
}

# Caso de sucesso
Response: { "statusCode": 200, body: {} }

# Caso de erro
Response: { "statusCode": 400, body: { "msg": 'alguma msg' } }

# Entidade
trade, user, inventory, inventorypokemon, pokemon
```

Tela de troca

PokeTroca

← PokeTroca

Telas de troca

Você deseja Em troca de **Fazer troca**

Inventário de teste6@teste.com	Seu inventário
tauros	jigglypuff
golduck	kabutops
fearow	gyarados
charmander	venomoth
weepinbell	gengar
charizard	electabuzz

Dados para Teste

Dados para realizar Login:


e-mail: dev


senha: dev

Ao realizar login, na tela principal uma lista de usuários será exibida. Digite então o e-mail do usuário para acessar o inventário e ir a tela de troca.

Treinadores Pokemon

Buscar inventário

 teste6@teste.com

 teste1@teste.com


Na tela de troca digite o nome do pokemon que deseja enviar para troca, o nome do pokemon que deseja em troca e selecione o botão "Fazer troca".


Telas de troca


Fazer troca


Inventário de teste6@teste.com

Seu inventário

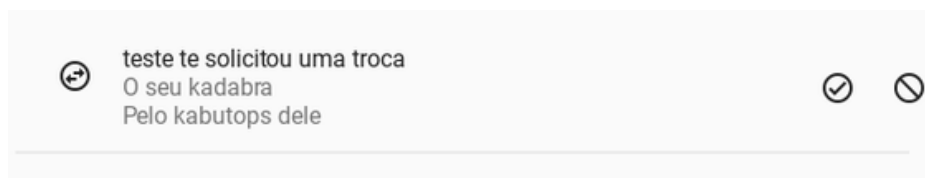
 tauros

 golduck

 jigglypuff

 kabutops

Ao finalizar a solicitação, volte para a tela principal e clique no ícone central escrito Trocas. Ao acessar essa tela é possível ver todas as solicitações de trocas disponíveis para o usuário que estamos utilizando nesse momento (no caso dev). Além de listar as solicitações é possível aceitar uma troca ou recusa-lá, caso seja aceito todo o método de troca é executado e finalmente foi realizado a troca do seu pokemon.



Na tela de inventário é possível ver todos os pokemons que você possui atualmente, inclusive observar se a troca foi realmente concluída.

	jigglypuff
	kabutops
	gyarados
	venomoth
	gengar

É possível também caso você queira, realizar o cadastro de um usuário novo ao invés de utilizar os usuários de teste.

CADASTRO

Nome

Nome de usuário

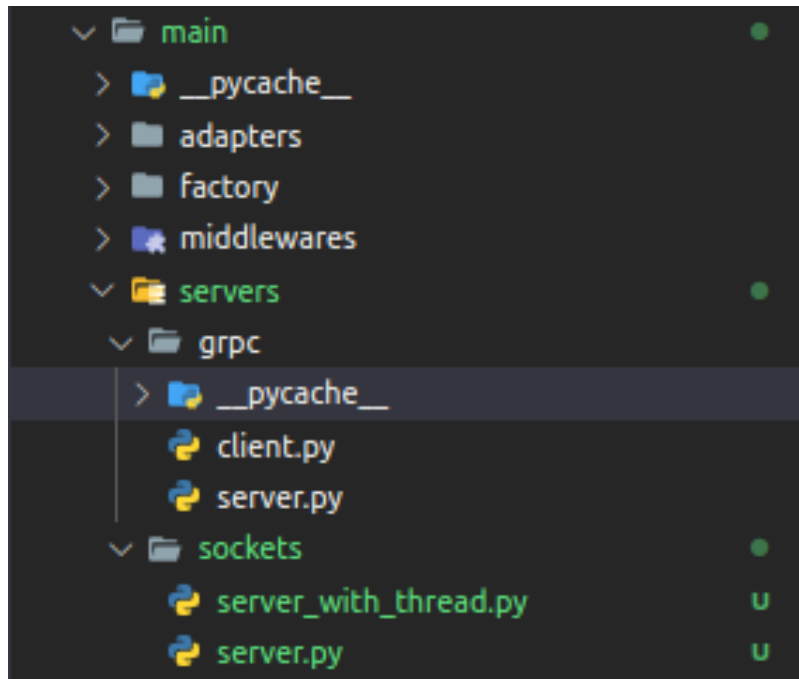
Senha

Cadastrar

Troca da comunicação de Socket para GRPC

Após realizado a implementação do projeto PokeTroca enfrentando todos os desafios provindos da utilização da API de Sockets, a quarta parte do Trabalho nós permitiu utilizar um middleware RMI que vinha com o intuito de facilitar a comunicação entre os processos, já que o middleware traz toda a parte da comunicação pronta, sendo necessário apenas aprender sobre a forma de como se utilizar o middleware que no caso foi o GRPC. Devido ao fato da utilização de uma Arquitetura bem consolidada, onde a ideia estava justamente no desacoplamento de cada um dos casos de usos apresentados anteriormente, inserir o GRPC no projeto foi algo bem tranquilo, A escolha inicialmente da Arquitetura Limpa como a arquitetura lógica do projeto, foi visando justamente os próximos passos, onde todo trabalho gasto na primeira implementação seria recompensado no momento em que o serviço de comunicação precisasse ser alterado. Portanto a maior dificuldade encontrada durante a quarta parte do trabalho estava justamente na forma de se utilizar o middleware GRPC e em como transformar a representação externa de dados utilizada anteriormente (JSON) para a utilizada pelo PROTOBUFFER (Mensagem).

Do lado do servidor, um novo diretório foi criado na camada Main, para guardar as implementações de cada um dos serviços, como representado na imagem a seguir:



Dessa forma, a implementação do GRPC é apenas um serviço adicional no projeto, possibilitando continuar utilizando a API de Sockets, caso deseje. Para executarmos então o novo servidor implementado, é preciso executar o comando a seguir:

```
python3 -m src.main.servers.grpc.server
```

A partir da execução do comando, o servidor GRPC estará funcionando e aguardando algum cliente se comunicar e utilizar a invocação remota dos metodos. Segue a implementação do servidor GRPC:

```

1 from concurrent.futures import ThreadPoolExecutor
2 from concurrent import futures
3 import time
4
5 import grpc
6 from src.main.middlewares.grpc_middlewares.protos.user.add_user_account_proto import add_user_account_pb2
7 from src.main.middlewares.grpc_middlewares.protos.user.add_user_account_proto import add_user_account_pb2_grpc
8 from src.main.middlewares.grpc_middlewares.services.user.add_user_account_service.add_user_account import AddUserAccountService
9 #import authenticate
10 from src.main.middlewares.grpc_middlewares.protos.user.authenticate_user_proto import authenticate_user_pb2
11 from src.main.middlewares.grpc_middlewares.protos.user.authenticate_user_proto import authenticate_user_pb2_grpc
12 from src.main.middlewares.grpc_middlewares.services.user.authenticate_user_service.authenticate_user import AuthenticateUserService
13 #import listUserAccount
14 from src.main.middlewares.grpc_middlewares.protos.user.list_user_account_proto import list_user_account_pb2
15 from src.main.middlewares.grpc_middlewares.protos.user.list_user_account_proto import list_user_account_pb2_grpc
16 from src.main.middlewares.grpc_middlewares.services.user.list_user_account_service.list_user_account import ListUserAccountService
17 #import listUserInventory
18 from src.main.middlewares.grpc_middlewares.protos.user.list_user_inventory_proto import list_user_inventory_pb2
19 from src.main.middlewares.grpc_middlewares.protos.user.list_user_inventory_proto import list_user_inventory_pb2_grpc
20 from src.main.middlewares.grpc_middlewares.services.user.list_user_inventory_service.list_user_inventory import ListUserInventoryService
21 #import AddTradeSolicitations
22 from src.main.middlewares.grpc_middlewares.protos.trade.add_trade_solicitations_proto import add_trade_solicitations_pb2
23 from src.main.middlewares.grpc_middlewares.protos.trade.add_trade_solicitations_proto import add_trade_solicitations_pb2_grpc
24 from src.main.middlewares.grpc_middlewares.services.trade.add_trade_solicitations_service.add_trade_solicitations import AddTradeSolicitationsService
25 #import ListTradeSolicitations
26 from src.main.middlewares.grpc_middlewares.protos.trade.list_trade_solicitations_proto import list_trade_solicitations_pb2
27 from src.main.middlewares.grpc_middlewares.protos.trade.list_trade_solicitations_proto import list_trade_solicitations_pb2_grpc
28 from src.main.middlewares.grpc_middlewares.services.trade.list_trade_solicitations_service.list_trade_solicitations import ListTradeSolicitationsService
29 #import RefuseExchange
30 from src.main.middlewares.grpc_middlewares.protos.trade.refuse_exchange_proto import refuse_exchange_pb2
31 from src.main.middlewares.grpc_middlewares.protos.trade.refuse_exchange_proto import refuse_exchange_pb2_grpc
32 from src.main.middlewares.grpc_middlewares.services.trade.refuse_exchange_service.refuse_exchange import RefuseExchangeService
33 #import TradePokemon
34 from src.main.middlewares.grpc_middlewares.protos.trade.trade_pokemon_proto import trade_pokemon_pb2
35 from src.main.middlewares.grpc_middlewares.protos.trade.trade_pokemon_proto import trade_pokemon_pb2_grpc
36 from src.main.middlewares.grpc_middlewares.services.trade.trade_pokemon_service.trade_pokemon import TradePokemonService
37
38 def serve():
39     server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
40     # add methods
41     add_user_account_pb2_grpc.add_AddUserAccountService(server, AddUserAccountService(), server)
42     authenticate_user_pb2_grpc.add_AuthenticateUserService(server, AuthenticateUserService(), server)
43     list_user_account_pb2_grpc.add_ListUserAccountService(server, ListUserAccountService(), server)
44     list_user_inventory_pb2_grpc.add_ListUserInventoryService(server, ListUserInventoryService(), server)
45     add_trade_solicitations_pb2_grpc.add_AddTradeSolicitationsService(server, AddTradeSolicitationsService(), server)
46     list_trade_solicitations_pb2_grpc.add_ListTradeSolicitationsService(server, ListTradeSolicitationsService(), server)
47     refuse_exchange_pb2_grpc.add_RefuseExchangeService(server, RefuseExchangeService(), server)
48     trade_pokemon_pb2_grpc.add_TradePokemonService(server, TradePokemonService(), server)
49
50     server.add_insecure_port('localhost:50052')
51     server.start()
52     server.wait_for_termination()
53
54 if __name__ == '__main__':
55     serve()

```

Já do lado do Cliente, houve uma modificação maior, pois agora o cliente deve escolher qual método irá executar de acordo com a funcionalidade que deseja cumprir. Portanto a parte do servidor que definia qual método seria executado, foi deslocado para o cliente, como no código a seguir:

```

16 from src.main.middlewares.grpc_middlewares.protos.trade.list_trade_solicitations_proto import list_trade_solicitations_pb2
17 from src.main.middlewares.grpc_middlewares.protos.trade.list_trade_solicitations_proto import list_trade_solicitations_pb2_grpc
18 #RefuseExchange
19 from src.main.middlewares.grpc_middlewares.protos.trade.refuse_exchange_proto import refuse_exchange_pb2
20 from src.main.middlewares.grpc_middlewares.protos.trade.refuse_exchange_proto import refuse_exchange_pb2_grpc
21 #TradePokemon
22 from src.main.middlewares.grpc_middlewares.protos.trade.trade_pokemon_proto import trade_pokemon_pb2
23 from src.main.middlewares.grpc_middlewares.protos.trade.trade_pokemon_proto import trade_pokemon_pb2_grpc
24 import grpc
25
26 def run():
27     with grpc.insecure_channel('localhost:50052') as channel:
28         route = 'user/inventory'
29         if (route == 'login'):
30             stub = authenticate_user_pb2_grpc.AuthenticateUserStub(channel)
31             request = authenticate_user_pb2.Request(username = "teste82@teste.com", password = "teste123")
32             response = stub.makeAuthenticateUserFactory(request)
33             print('RESPONSE', response)
34             return response
35         if (route == 'user/create'):
36             stub = add_user_account_pb2_grpc.AddUserAccountStub(channel)
37             request = add_user_account_pb2.Request(username = "teste81@teste.com", password = "teste123", name = "Pedro Oliveira")
38             response = stub.makeUserAccountFactory(request)
39             print('RESPONSE', response)
40             return response
41         if (route == 'get-user'):
42             stub = list_user_account_pb2_grpc.ListUserAccountStub(channel)
43             request = list_user_account_pb2.Request()
44             response = stub.makeListUserAccountFactory(request)
45             print('RESPONSE', response)
46             return response
47         if (route == 'user/inventory'):
48             stub = list_user_inventory_pb2_grpc.ListUserInventoryStub(channel)
49             request = list_user_inventory_pb2.Request(id = 4)
50             response = stub.makeListUserInventoryFactory(request)
51             print('RESPONSE', response)
52             return response

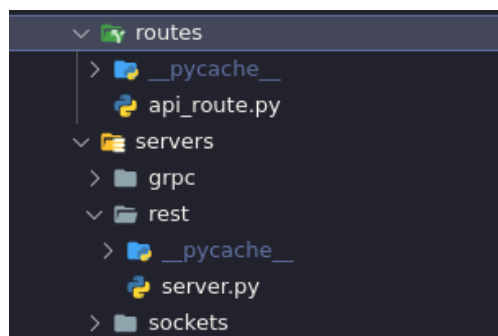
```

Utilizando comunicação REST para o PokeTroca

Após realizadas as outras partes do trabalho, onde utilizamos o método de invocação remota e de sockets, foi a vez da implementação utilizando API REST. Esse estilo de projeto de arquitetura de desenvolvimento web é amplamente utilizada hoje em inúmeras empresas por ser altamente escalável, possibilitar que serviços utilizem outros serviços, performar bem e são desenvolvidas para serem facilmente administradas através de requisições HTTP, indicando sua ação (GET, POST, PUT, PATCH E DELETE, etc.).

Para o desenvolvimento, foi utilizado o Flask para o servidor. o Flask é um micro-framework multiplataforma que provê um modelo simples para o desenvolvimento web que, apesar de ser bem extensível, é destinado principalmente para pequenas aplicações com requisitos mais simples como o nosso caso. Os benefícios "giram" em torno de simplicidade, rapidez no desenvolvimento, e uma boa performance.

As alterações necessárias no servidor foram, além de incluir o flask nas dependências do projeto, incluir mais uma camada de comunicação, da mesma forma que foi feita para a parte de GRPC e incluir uma parte para a definição das rotas que poderão ser utilizadas:



Para definir as rotas é necessário fazer a configuração da seguinte forma, onde você "recebe" a rota nos parâmetros da URL e despacha a operação correta a ser realizada:

```
@api_routes_bp.route("/api/get-users", methods=["GET"])
def getAllUsers():
    result = ListUserAccountFactory.makeListUserAccountFactory(request.json["params"])
    return jsonify(result)

@api_routes_bp.route("/api/user/inventory", methods=["GET"])
def getUserInventory():
    result = ListUserInventoryFactory.makeListUserInventoryFactory(request.json["params"])
    return jsonify(result)

@api_routes_bp.route("/api/user/create", methods=["POST"])
def createUser():
    result = AddUserAccountFactory.makeUserAccountFactory(request.json["params"])
    return jsonify(result)

@api_routes_bp.route("/api/login", methods=["GET"])
def getUser():
    result = AuthenticateUserFactory.makeAuthenticateUserFactory(request.json["params"])
    return jsonify(result)

# Trade Routes
@api_routes_bp.route("/api/trade/solicitations", methods=["GET"])
def getTradeSolicitation():
    result = ListTradeSolicitationsFactory.makeListTradeSolicitationsFactory(request.json["params"])
    return jsonify(result)

@api_routes_bp.route("/api/trade/create", methods=["POST"])
def createTradeSolicitation():
    result = AddTradeSolicitationsFactory.makeAddTradeSolicitationsFactory(request.json["params"])
    return jsonify(result)
```

A requisição é retornada no formato Json e, por isso, foram poucas as modificações necessárias para ambos os lados partindo da parte implementada em sockets. Uma outra diferença foi, também, a criação do arquivo que provê os serviços através de uma API Flask:

```
1 from flask import Flask
2 from flask_cors import CORS
3 from src.main.routes.api_route import api_routes_bp
4
5 app = Flask(__name__)
6 CORS(app)
7
8 app.register_blueprint(api_routes_bp)
9
10 app.run(host="0.0.0.0", port=5000)
```

Já do lado do Cliente, foi utilizada a biblioteca requests do python, que permite fazer requisições HTTP utilizando todas as ações possíveis e já converter a resposta em Json. Logo, além de incluir a biblioteca nas dependências do projeto cliente, como centralizamos as comunicações do cliente em apenas um arquivo, foi necessário apenas fazer modificações no mesmo, como segue abaixo:

```
import requests

You, yesterday | 1 author (You)
class RestRequests():
    def __init__(self, **kwargs):
        super(RestRequests, self).__init__(**kwargs)

    #Este será o método genérico de requisição
    def req(self, type, route, params):
        api_url = f"http://localhost:5000/api/{route}"
        jsonData = {
            "params": params,
        }
        response = {}
        if(type == "POST"):
            response = requests.post(api_url, json=jsonData).json()

        if(type == "PUT"):
            response = requests.put(api_url, json=jsonData).json()
        return response
```

Desta forma, tivemos que modificar os locais onde utilizávamos o método req() para que enviassem qual método seria utilizado na requisição, se era GET, POST ou PUT e tratamos isso nesse arquivo que realiza as requisições. Além disso utilizamos a url padrão atribuída na variável api_url, que é http://localhost:5000/api/nome-da-rota. A parte da url que representa o nome da rota é que diferencia qual operação o servidor deve realizar.

Para a execução do trabalho você precisa digitar no terminal os seguintes comandos para o cliente e para o servidor em seus respectivos diretórios:

```
make setup
```

Para que as novas dependências sejam também instaladas.

```
make run
```

Para rodar tanto o servidor quanto o cliente. O servidor naturalmente deve ser iniciado primeiro que o cliente.

Conclusão

Por fim, ao realizar a confecção do trabalho, as dificuldades não foram muito grandes devido a dupla já ter contato com API's Restful e conhecer como a mesma funciona, independentemente da tecnologia. Logo, as dificuldades foram em torno de como utilizar em python, já que é a primeira vez utilizando a linguagem para ambos. Além disso, a maneira como implementamos o trabalho desde a parte de sockets permitiu que as mudanças ao longo das demais partes não fossem críticas e problemáticas. Em relação ao trabalho de GRPC, foi bem mais tranquilo a implementação em Rest porque GRPC foi algo totalmente novo para nós mas ainda assim não foi impedimento para a implementação. Em relação as demais partes, a impressão deixada foi que a implementação Rest conseguiu um desempenho melhor, deixando a navegação no cliente mais fluída, seguido de sockets e GRPC, ele foi o melhor. Enfim, a realização do trabalho como um todo foi de grande valia já que foi uma maneira de não nos engessar ao que já sabemos e conhecemos sobre comunicação em sistemas distribuídos. É de suma importância poder sempre ter uma visão diferente quanto as possibilidades e contextos possíveis em um projeto levando os casos vistos em aula em consideração.

UFV

Universidade Federal de Viçosa



Versão 4.0